1296287

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

March 15, 2005

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE.

APPLICATION NUMBER: ***60/546,194***
FILING DATE: ***February 23, 2004***
RELATED PCT APPLICATION NUMBER: ***PCT/US05/05481***

UNITED STATES PATENT AND TRADEMARK OFFICE DEPARTMENT OF COMMERCE

Certified by

[illegible]

Under Secretary of Commerce
for Intellectual Property
and Director of the United States
Patent and Trademark Office

13281 U.S.PTO
022304

15535 U.S.PTO
60/546194
022304

PTO/SB/16 (08-03)
Approved for use through 07/31/2006. OMB 0651-0032
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53 (c).

INVENTOR(S)

Given Name (first and middle [if any])	Family Name or Surname	Residence (City and either State or Foreign Country)
Nicolas	POPP	Menlo Park, California

☐ *Additional inventors are being named on the _____ separately numbered sheets attached hereto*

TITLE OF THE INVENTION (500 characters max)

TOKEN AUTHENTICATION SYSTEM

Direct all correspondence **CORRESPONDENCE ADDRESS**

☒ Customer Number **23838**

OR

☐ Firm *or* Individual Name					
Address					
Address					
City		State		ZIP	
Country		Telephone		Fax	

ENCLOSED APPLICATION PARTS *(check all that apply)*

☒ Specification *Number of Pages* 8 ☐ CD(s), Number

☐ Drawing(s) *Number of Sheets* ☐ Other (specify)

☐ Application Data Sheet. See 37 CFR 1.76

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT

☐ Applicant claims small entity status. See 37 CFR 1.27.

☐ A check or money order is enclosed to cover the filing fees

☒ The Director is hereby authorized to charge filing fees or credit any overpayment to Deposit Account Number: 11-0600

☐ Payment by credit card. Form PTO-2038 is attached.

FILING FEE AMOUNT ($) 160.00

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

☒ No.

☐ Yes, the name of the U.S. Government agency and the Government contract number are: _____.

[Page 1 of 1]

Respectfully submitted,
SIGNATURE [illegible]

Date February 23, 2004

TYPED or PRINTED NAME Gary S. Morris

REGISTRATION NO. (*if appropriate*) 40,735

TELEPHONE 202/220/4200

Docket Number: 12832/100004

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

Token Authentication System

An embodiment of the present invention includes a protocol for generating One Time Passwords ("OTPs") that can be used to authenticate a user. The OTPs are generated by a token, which can be a physical device that includes mechanisms to prevent the unauthorized modification or disclosure of the software and information that it contains, and to help ensure its proper functioning.

An embodiment of this protocol can be sequence based, and can be two-factor, e.g., based upon something the user knows (such as a Personal Identification Number and a physical device that the user posesses (e.g., a token.)

The protocol for generating the OTPs can be based upon a counter, e.g., a monotonically increasing number based, for example, on the number of times a OTP has been requested from the token. The value of the OTP can be displayed on a token, and can be easily read and entered by the user, e.g., via a keyboard coupled to a computer that is in turn coupled to a network. The OTP can be transportable over the RADIUS system.

An embodiment of the protocol in accordance with the present invention can based on an increasing counter value and a static symmetric key known only to the token and an authentication service (the "strong auth" service.) An OTP value can be created using the HMAC-SHA1 algorithm as defined in RFC 2104. This hashed MAC algorithm has the strength of SHA-1, but allows the addition of a secret during the calculation of the output.

The output of the HMAC-SHA1 calculation is 160 bits. However, this value can be truncated to a length that can be easily entered by a user. Thus,

OTP = Truncate(HMAC-SHA1 (Count, Secret))

Both the client and authentication server can calculate the OTP value. If the value received by the server matches the value calculated by the server, then the user can be said to be authenticated. Once an authentication occurs at the server, the server increments the counter value by one.

Although the servers counter value can be incremented only after a successful OTP authentication, the counter on the token can be incremented every time the button is pushed. Because of this, the counter values on the server and on the token will be often out of synchronization. Indeed, there is a good chance that the token will always be out of synchronization with the server given the user environment (e.g., the user pushes the button unnecessarily, button is pushed accidentally, user mistypes the OTP, etc.)

Because the server's counter will only increment when a valid OTP is presented, the server's counter value on the token is expected to always be less than the counter value on token. The key to resynchronization is to ensure it's not a burden to either the user or the enterprise IT department.

Synchronization of counters is in this scenario can be achieved by having the server calculate the next *n* OTP values and determine if there is a match, where *n* is an integer. If we assume that the difference between the count at the token and the count at the server is 50, then depending on the implementation of the strong auth server, the server would at most have to calculate the next 50 OTP values. thus, for example, if the correct OTP is found at the *n+12* value, then the server can authenticate the user and then increment the counter by 12.

It is important to carefully choose a value for *n* that can be easily calculated by the server. There should be upper bounds to ensure the server doesn't check OTP values forever, thereby succumbing to a Denial of Service attack.

Truncating the HMAC-SHA1 value to a 6 character value could make a brute force attack easy to mount, especially if only numeric digits are used. Because of this, such attacks can be detected and stopped at the strong auth server. Each time the server calculates an OTP that does not validate, it should record this and implement measures to prevent being swamped, e.g., at some point, turn away future requests for validation from the same source. Alternatively, the user can be forced to wait for a longer period of time between validation attempts.

Once a user is locked out, the user can be to "self unlock" by providing a web interface that would require the user, for example, to enter multiple OTP in a sequence, thus proving they have the token.

Once the shared secret has been combined with the counter, a 160 bit (20-byte) HMAC-SHA1 result can be obtained. In one embodiment, at most four bytes of this information for our OTP. The HMAC RFC (RFC 2104 - HMAC: Keyed-Hashing for Message Authentication) further warns that we should use at least half the HMAC result in Section 5 **Truncated Output**:

> Applications of HMAC can choose to truncate the output of HMAC by outputting the t leftmost bits of the HMAC computation ... We recommend that the output length t be not less than half the length of the hash output ... and not less than 80 bits (a suitable lower bound on the number of bits that need to be predicted by an attacker).

Thus, another way is needed to choose only four or fewer bytes of the HMAC result in a way that will not weaken either HMAC or SHA1. In one *Dynamic offset truncation*, described below, can be used.

Dynamic offset truncation.

The purpose of this technique is to extract a four byte dynamic binary code from an HMAC-SHA1 result while still keeping most of the cryptographic strength of the MAC.

1. Take the last byte (byte 19)
2. Mask off the low order four bits as the offset value
3. use the offset value to index into the bytes of the HMAC-SHA1 result.
4. return the following four bytes as the dynamic binary code.

The following code example describes the extraction of a dynamic binary code given that `hmac_result` is a byte array with the HMAC-SHA1 result:

```
    int offset   =  hmac_result[19] & 0xf ;
    int bin_code = (hmac_result[offset]  & 0x7f) << 24
                 | (hmac_result[offset+1] & 0xff) << 16
                 | (hmac_result[offset+2] & 0xff) <<  8
                 | (hmac_result[offset+3] & 0xff) ;
```

The following is an example of using this technique:

SHA-1 Hash bytes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1f	86	98	69	0e	02	ca	16	61	85	50	ef	7f	19	da	8e	94	5b	55	5a

1. The last byte (**byte 19**) has the hex value **0x5a**.
2. The value of the lower four bits is 0xa (the offset value).
3. The offset value is byte **10** (0xa).
4. The value of the four bytes starting at byte **10** is **0x50ef7f19** which is the dynamic binary code.

We treat the dynamic binary code as a 31 bit, unsigned, big-endian integer; the first byte is masked with a 0x7f.

The One Time Password for a given secret and moving factor can vary based on three parameters: **Encoding Base**, **Code Digits**, and **Has Checksum**. With `10` as an **Encoding Base**, 7 **Code Digits** and Has Checksum set to TRUE, we continue with the above example:

- The hex value 0x50ef7f19 converts to 1357872921 base 10.
- The last 7 digits are 7872921
- The credit card checksum of the code digits calculates 7 as the checksum.
  $10 - ((5 + 8 + 5 + 2 + 9 + 2 + 2) \bmod 10) = 10 - (33 \bmod 10) = 10 - 3 = 7$
- Yielding the following OTP: 78729217

The following is the Glossary:

Checksum Digit – Result of applying the Credit-Card checksum algorithm to the Code Digits. This digit is optional. This check should catch any single transposition or any single character mistyped.

Code Digits – This parameter indicates is the number of digits to be obtained from the binary code. We calculate the **Code Digits** by converting the **Binary Code** to the **Encoding Base**, zero padding to be at least **Code Digits**, and taking the right hand (low order) digits. With `ten` as an **Encoding Base**, no more than `nine` **Code Digits** can be supported by the 31-bit **Binary Code**.

Credit Card checksum algorithm – This checksum algorithm has the advantage that it will catch any single mistyped digit, or any single adjacent transposition of two digits. These are the most common types of user errors.

Encoding Base – This parameter indicates what base to use for the password. Base 10 is the preferred base because it can be entered in a numeric pad.

Has Checksum Digit – This boolean parameter indicates if a checksum digit should be added to create the **OTP**. If there is no checksum digit, just the **Code Digits** are used as the **OTP**. If there is a checksum digit, it is calculated using the **Code Digits** as input to the **Credit-Card checksum algorithm**.

OTP – The One-Time-Password. This value is constructed by appending the **Checksum Digit**, if any, to the right of the **Code Digits**. If there is no **Checksum Digit**, the **OTP** is the same as the **Code Digits**.

One Time Number value is calculated by first shifting in the Hash Bits, and then the Synchronization Bits, and then shifting in the result of the `check_function()` of the prior intermediate value.

The binary value is then translated to appropriate character values. There are three likely character representations of the password: Decimal, Hexadecimal, and Alpha-Numeric.

	Decimal	**Hexadecimal**	**Alpha-Numeric**
Required Display Type	7-Segment	7-Segment	Alpha-Numeric
Conversion Cost	Moderate	Trivial	Simple
Bits per character	3.2	4	5
Keyboard Entry	Simple	Easy	Easy
Cell Phone Entry	Easy	Difficult	Difficult

Decimal.

The token can convert the dynamic binary code to decimal and then display then last Code Digits plus optionally the checksum. For example, for a six digit, no checksum, decimal token will convert the binary code to decimal and display the last six digits.

Hexadecimal.

The token can convert the dynamic binary code to hexadecimal and then display then last Code Digits plus optionally the checksum. For example, for a six digit, no checksum, hexadecimal token will convert the binary code to hexadecimal and display the last six digits.

Alphanumeric.

The token can convert the dynamic binary code to base 32 and then display then last Code Digits plus optionally the checksum. For example, for a six digit, no checksum, base 32 token will convert the binary code to base 32 and display the last six digits.

The PIN.

In order to be a true two-factor authentication token, there can also be a "what you know" value in addition to the "what you have" value of the OTP. This value is usually a static PIN or password known only to the user. This section discusses three alternative architectures to validate this static PIN in accordance with the present invention.

PIN Validation in the Cloud.

Enabling PIN validation in the cloud can bind a single PIN to a single token. The PIN should be protected over the wire to ensure its not revealed. PIN management (set, change) should be implemented by the strong auth service.

PIN Validation at the Enterprise.

Enabling PIN validation in the cloud may mean multiple PIN's for a single token. The PIN need never leave the "security world" of the enterprise. PIN management can be implemented by the enterprise.

PIN Validation at the Token.

This embodiment can be implemented using a keypad on the token. The PIN is never sent over the wire, and may be used as a seed to the OTP algorithm. The PIN may be used to simply unlock the token.

Example Data Flows.

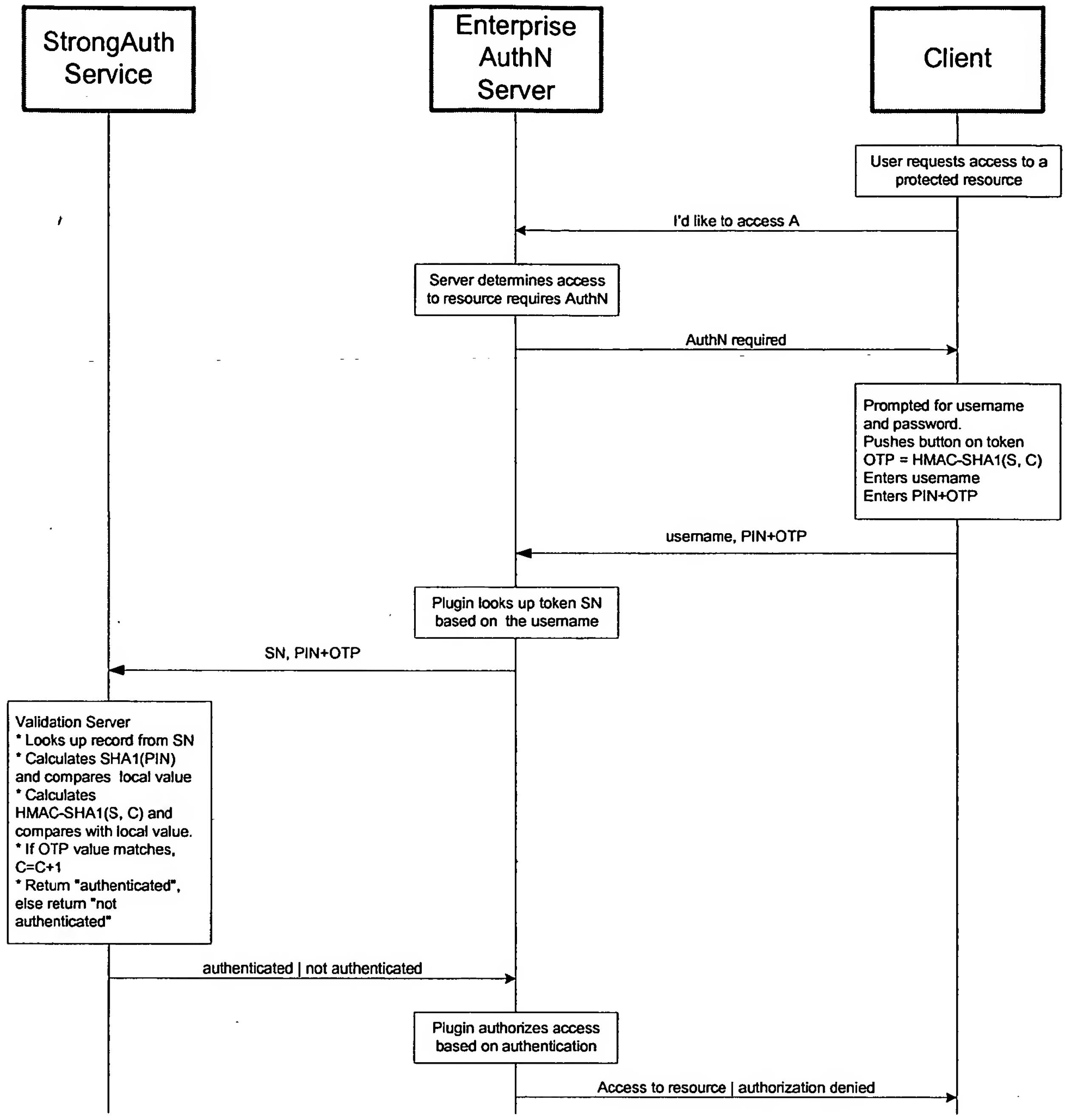
This section describes two data flow scenarios. The first scenario assumes that the StrongAuth service does not knows the username associated with a token. In this scenario it is assumed that the PIN management operations require the user to enter their token SN, instead of their user name.

The second scenario is very similar to the first, except by the fact that the user name is the key into the StrongAuth service, instead of the token SN. There are a few issues in this case. First, the username is known to the service, and second we must come up with a scheme to ensure a unique mapping from a username to a token SN

Data Flow #1

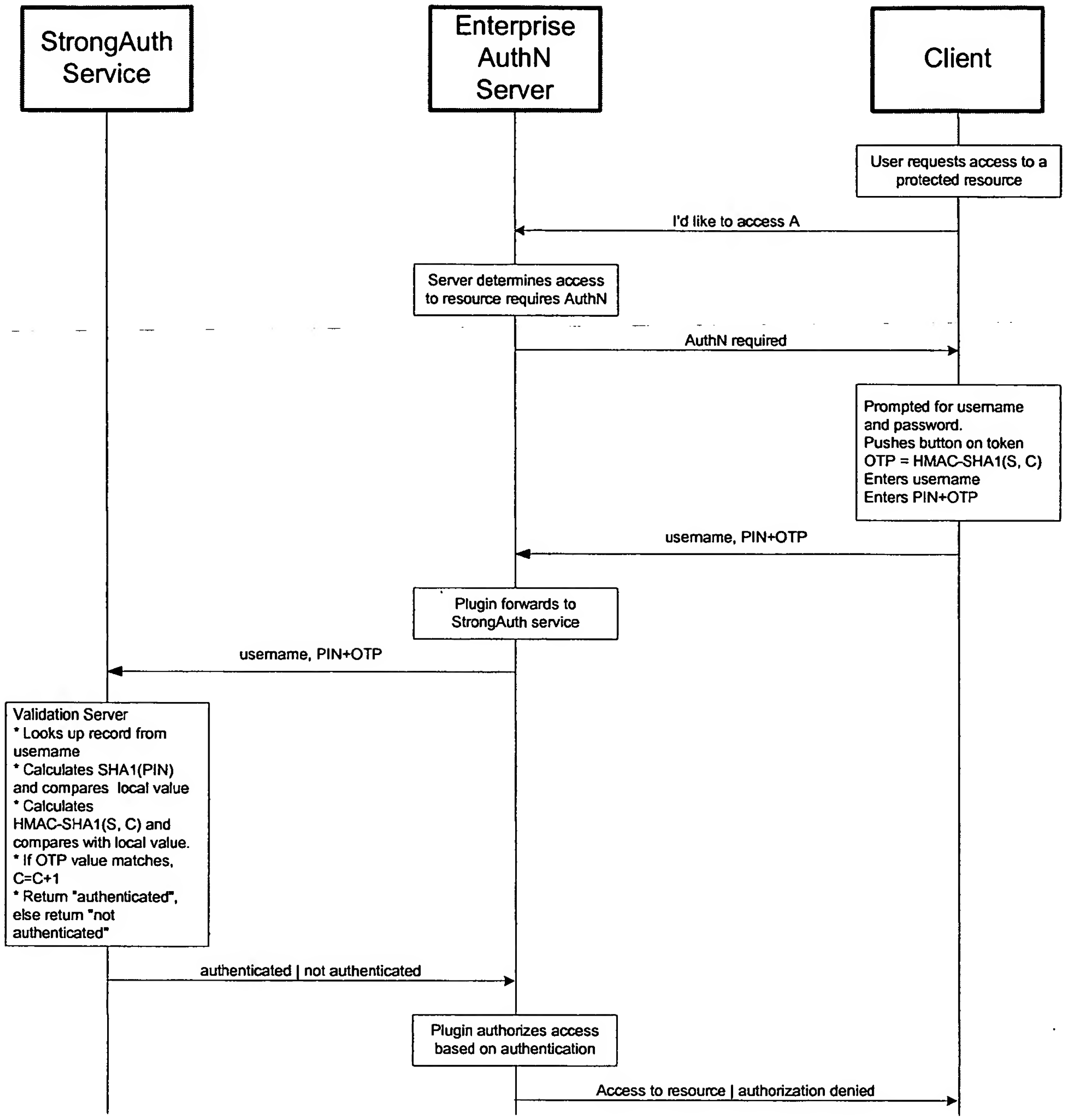
Assumptions:

- Token SN and S distributed to StrongAuth Service
- Customer account associated with each token
- PIN associated with a particular token
- StrongAuth service saves only the SHA-1 hash of the PIN in the DB
- Token SN added as an attribute to the user account at the Enterprise, the plugin maps the username to the SN.


StrongAuth Service
Enterprise AuthN Server
Client
User requests access to a protected resource
I'd like to access A
Server determines access to resource requires AuthN
AuthN required
Prompted for username and password.
Pushes button on token
OTP = HMAC-SHA1(S, C)
Enters username
Enters PIN+OTP
username, PIN+OTP
Plugin looks up token SN based on the username
SN, PIN+OTP
Validation Server
* Looks up record from SN
* Calculates SHA1(PIN) and compares local value
* Calculates HMAC-SHA1(S, C) and compares with local value.
* If OTP value matches, C=C+1
* Return "authenticated", else return "not authenticated"
authenticated | not authenticated
Plugin authorizes access based on authentication
Access to resource | authorization denied

Data Flow #2.
Assumptions:

- Token SN and S distributed to StrongAuth Service
- Customer account and username is associated with each token
- PIN associated with a particular token
- StrongAuth service saves only the SHA-1 hash of the PIN in the DB
- Username is forwarded from enterprise to the Strong Auth service.


StrongAuth Service
Enterprise AuthN Server
Client
User requests access to a protected resource
I'd like to access A
Server determines access to resource requires AuthN
AuthN required
Prompted for username and password.
Pushes button on token
OTP = HMAC-SHA1(S, C)
Enters username
Enters PIN+OTP
username, PIN+OTP
Plugin forwards to StrongAuth service
username, PIN+OTP
Validation Server
* Looks up record from username
* Calculates SHA1(PIN) and compares local value
* Calculates HMAC-SHA1(S, C) and compares with local value.
* If OTP value matches, C=C+1
* Return "authenticated", else return "not authenticated"
authenticated | not authenticated
Plugin authorizes access based on authentication
Access to resource | authorization denied

What is claimed is:

1. A method for calculating a One Time Password, comprising:

concatenting a secret with a count, where the secret is uniquely assigned to a token and is shared between the token and an authentication server, and the count is a number that increases monotonically at the token with the number of One Time Passwords generated at the token and increases monotonically at the authentication server with each calculation of a One Time Password at the authentication server;

calculating HMAC-SHA1 based upon the concatenated secret and count; and

truncating the result of the HMAC-SHA1 calculation to obtain a One Time Password.